COSC 404
Database System Implementation

Recovery

Dr. Ramon Lawrence
University of British Columbia Okanagan
ramon.lawrence@ubc.ca

Recovery Motivation

A database system like any computer system is subject to various types of failures.

The database system must ensure the ACID properties (specifically durability and atomicity) despite failures.

We will categorize the various types of failures, and provide approaches for *recovering* from failures.

The process of restoring the database to a consistent state after a failure is called ***recovery**,* and is performed by the ***recovery system***.

Why is Recoverability Needed?

Recoverability is needed because the database system can fail for many reasons during transaction processing:

- **Computer Failure** - computer crash due to hardware, software, or network problems.
- **Disk Failure** - disk fails to correctly read/write blocks
- **Physical Problems/Catastrophes** - external problems resulting in data loss or system destruction (e.g. earthquake)

Transaction failures (but not database system failures):

- **Transaction Error** - error in transaction (e.g. divide by 0)
- **Exception Conditions** - transaction detects exception condition (e.g. data not present, insufficient bank funds)
- **Concurrency Control Enforcement** - transaction can be forced to abort to resolve deadlock or for serializability.

Failure Classification

The various types of failures can be classified in three categories:

- Transaction Failures:
  - **Logical errors**: Transaction cannot complete due to some internal error condition (bad input, data not found).
  - **System errors**: The database system must terminate an active transaction due to an error condition (e.g. deadlock).
- Software Failures:
  - **System crash**: A failure causes the system to crash, but non-volatile storage contents are not corrupted.
  - Examples: software design errors, bugs, buffer/stack overflows
- Hardware Failures:
  - **Disk failure**: A head crash destroys all or part of disk storage.
  - Examples: overutilization/overloading (used beyond its design), wearout failure, poor manufacturing

Terminology

A system is ***reliable*** if it functions as per specifications and produces a correct output for a given input.

A system ***failure*** occurs if it does not function according to specifications and fails to deliver the service desired.

An ***error*** occurs if the system assumes an undesirable state.

A ***fault*** is detected when either an error is propagated from one component to another or the failure of a component is detected.

Reliability Mechanisms

Fault Avoidance

- Attempt to eliminate all forms of hardware and software errors.

Fault Tolerance

- Provide component redundancies that cater to faults occurring within the system and its components.

Tradeoff:

- Fault tolerance requires more components.
- More components means more faults.
- Therefore, more components are need to handle the increasing faults.

Storage Structure (review)

Volatile storage does not survive system crashes.
- main memory, cache memory

Nonvolatile storage survives system crashes.
- Hard drive, solid-state drive

Stable storage is a *theoretical* form of storage that survives all failures.
- Approximated by maintaining multiple copies on distinct nonvolatile media.
- Practically achieving stable storage requires duplication of information such as maintaining multiple copies of each block on separate disks (RAID), or sending copies to remote sites to protect against disasters such as fire or flooding.
  - e.g. Multiple availability zones with Amazon hosting

Data Access

Physical blocks are those blocks residing on the disk. ***Buffer blocks*** are the blocks residing temporarily in main memory.

Block movements between disk and main memory are initiated through the following two operations:
- **input**(*B*) transfers the physical block *B* to main memory.
- **output**(*B*) transfers the buffer block *B* to the disk.

Each transaction T_i has its private work area in which local copies of all data items accessed and updated by it are kept. Assume that T_i's local copy of a data item *X* is called x_i.

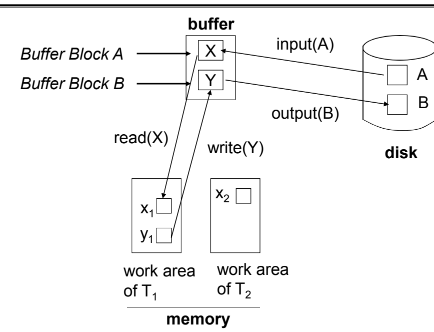
Data Access (2)

A transaction transfers data items between system buffer blocks and its private work-area using operations:
- **read**(X, x_i) assigns the value of item *X* to the local variable x_i.
- **write**(X, x_i) assigns the value of local variable x_i to data item *X* in the buffer block.
- Both these commands may require an **input**(B_X), if the block B_X in which *X* resides is not already in memory.

Transactions perform **read**(*X*) while accessing *X* for the first time; all subsequent accesses are to the local copy. After last access, transaction executes **write**(*X*).

output(B_X) need not immediately follow **write**(*X*). System can perform the **output** operation when it deems fit.

Example of Data Access



Buffer Management

The blocks in a database buffer are managed by a ***replacement policy*** (such as LRU).

Other considerations:
- **steal** vs. **no-steal** – no-steal prevents a buffer that is written by an uncommitted transaction to be saved to disk (removed from the buffer). Steal policy allows writing uncommitted updates.
  - Implemented using a pin bit on each buffer block.
- **force** vs. **no-force** – A force approach writes updates for committed transactions to disk immediately. No-force allows a committed update to remain in the buffer for some time.

Databases typically implement steal/no-force as it provides the most flexibility and best performance.

Log-Based Recovery

In log-based recovery, a ***log*** is kept on stable storage, and consists of a sequence of *log records*.

The log will record the sequence of database operations, and can be used to replay the database actions after a failure. The recovery manager uses the log to restore data items to their consistent state.

Recovery is related to concurrency control. We will assume that strict 2PL is performed that guarantees an item updated by a transaction T cannot be updated by another transaction until transaction T commits or aborts.

Log-Based Recovery Log Records

There are several types of log records:

- ***Start Records:*** When transaction T_i starts, it registers by writing a <T_i **start**> log record.
- ***Commit Records:*** When T_i finishes its last statement and successfully commits, the record <T_i **commit**> is written.
- ***Abort Records:*** When T_i aborts for whatever reason, the record <T_i **abort**> is written.
- ***Update Records:*** *Before* T_i executes **write**(X), a log record <T_i, X, V_1, V_2> is written, where V_1 is the value of X before the write, and V_2 is the value to be written to X.
  - That is, T_i has performed a write on data item X. X had value V_1 before the write, and will have value V_2 after the write.

Log records are written to stable storage.

Log Record Buffering

Log records are buffered in main memory, instead of being output directly to stable storage. Log records are output to stable storage when a block of log records in the buffer is full, or a **log force** operation is executed.

- Several log records can thus be output using a single output operation, reducing the I/O cost.

These rules must be followed if log records are buffered:

- Log records are output in the order in which they are created.
- Transaction T_i enters the commit state after the log record <T_i **commit**> has been output to stable storage.
- Before a block of data in main memory is output to the database, all log records pertaining to data in that block must have been output to stable storage. (This rule is called the **write-ahead logging** or **WAL** rule.)

Undo/Redo Logging

Undo/redo logging performs recovery by:

- **undo** updates for transactions that are not committed
- **redo** updates for transactions that were committed before failure

Redo/undo logging (WAL) rule:

- Before modifying any database element X on disk because of changes made by some transaction T, it is necessary that update record <T, X, V_1, V_2> appear on disk.

Write-Ahead Logging

Question: Write-ahead logging means:

A) If a data item is updated, it must be written to storage before the log record.

B) If a data item is read, it must read a written, committed value.

C) An updated data item must only be written to storage after the log record for the update is written to storage.

D) None of the above

Recovery with Undo/Redo Logging

The recovery system must:

- Redo all the committed transactions in the order earliest-first.
- Undo all uncompleted transactions in the order latest-first.

When the system recovers, it does the following:

- 1) Initialize *undo-list* and *redo-list* to empty.
- 2) **First pass**: Scan the log backwards from end to build list of transactions to undo and redo.
- 3) **Second pass**: Scan the log forwards from the beginning and redo updates of committed transactions.
- 4) **Third pass**: Scan the log backwards from end and undo updates of uncommitted transactions.
- 5) For each undo transaction T, write a <T **abort**> log record. Flush the log and resume normal operation.

Undo/Redo Recovery Example

The log as it appears at three instances of time:

(a)	(b)	(c)
<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 1000, 950>	<T_0, A, 1000, 950>	<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 700, 600>	<T_1, C, 700, 600>
		<T_1 commit>

Recovery actions in each case above are:

- (a) undo (T_0): B is restored to 2000 and A to 1000.
- (b) redo (T_0) and undo (T_1): A set to 950 and B set to 2050 then C is restored to 700.
- (c) redo (T_0) and redo (T_1): A and B are set to 950 and 2050 respectively. Then C is set to 600.

Undo/Redo Logging

Question: How many of the following statements are true?

- i) The first pass scans log forward to build undo and redo lists.
- ii) The second pass scans log forward performing redo.
- iii) The third pass scans log forward performing undo.
- iv) An update that is "redone" may or may not change the actual value in storage.

A) 0
B) 1
C) 2
D) 3
E) 4

Checkpoints

Recovery using the entire log would be expensive as the log grows in size over time.

To reduce the size of the log in order to make recovery faster, ***checkpoints*** are used to speed up recovery.

Checkpointing (blocking)

Checkpointing approach that blocks new transactions:

- 1) Stop accepting new transactions.
- 2) Wait until all currently running transactions either commit or abort.
- 3) Output all log records currently residing in main memory onto stable storage. (flush log) Output all updated buffers.
- 4) Write a log record <**checkpoint**> and flush log again.
- 5) Resume accepting transactions.

This guarantees all transactions before the checkpoint have their results reflected in the database. Recovery only needs to focus on log after the checkpoint.

Online (fuzzy) Checkpointing

The biggest problem with the previous technique is the system must stop processing transactions during the checkpoint.

Online checkpointing allows transactions to continue to run and be submitted during the procedure:

- 1) Write a log record <checkpoint start (T_1 ... T_N)> where T_1...T_N are the currently executing transactions. (flush log)
- 2) Write to disk all ***dirty*** buffers that have been modified before the checkpoint start. The buffers written include buffers changed by uncommitted transactions.
  - Note that the checkpoint procedure does not write dirty buffers that get modified between the checkpoint start and the checkpoint end records.
- 3) After all dirty buffers (recorded at checkpoint start) have been flushed, write a log record <checkpoint end> and flush the log.

Online Checkpointing

Question: How many of the following statements are true?

- i) Transactions may still run during an online checkpoint.
- ii) All updates in the buffer (committed or not) when the checkpoint starts are written to storage by end of checkpoint.
- iii) Updates in the buffer done after checkpoint start are written to storage.
- iv) The checkpoint start record contains all transactions, running and committed, before the checkpoint.

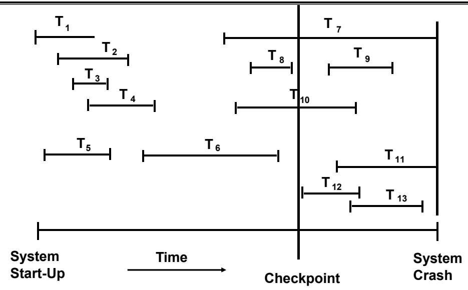
A) 0
B) 1
C) 2
D) 3
E) 4

Recovery using Undo/Redo and Checkpointing

Steps for recovery using undo/redo and checkpointing:

- 1) First pass backwards scan stops at the first start checkpoint log record found with a matching end checkpoint.
  - This scan will enumerate all transactions since last checkpoint and all active transactions when checkpoint began.
  - Divide these transactions into undo and redo lists.
- 2) Second pass forward scan starts at start checkpoint record and ends when all transactions are redone.
- 3) Third pass backwards scan starts at end of log and stops when all transactions in the undo list have been undone.
  - We know a transaction has no more operations when we encounter its transaction start log record.

Undo/Redo Checkpoints Example



What transactions are undone, redone, or committed?

Undo/Redo Recovery Example

The recovery algorithm on the following log:

<T_0 **start**>	**First Backwards Pass. (build lists from end)**
<T_0, A, 0, 10>	**Forwards Pass - Redo (start at checkpoint)**
<T_0 **commit**>	**Backwards Pass - Undo (start at end)**
<T_1 **start**>	Undo T1 complete. (Undo complete.)
<T_1, B, 0, 10>	Undo T1 write on B value now 0.
<T_2 **start**>	Undo T2 complete.
<T_2, C, 0, 10>	Undo T2 write on C value now 0.
<T_2, C, 10, 20>	Undo T2 write on C value now 10.
<checkpoint start (T_1, T_2)>	Checkpoint: T1, T2 were active (undo-list)
<checkpoint end>	
<T_3 **start**>	
<T_3, A, 10, 20>	Redo T3 write on A value now 20.
<T_3, D, 0, 10>	Redo T3 write on D value now 10.
<T_3 **commit**>	T3 in redo-list.
<T_1 abort>	Write abort transaction to log.
<T_2 abort>	Write abort transaction to log.

Undo/Redo Recovery with Checkpoints

Question: How many of the following statements are true?

- i) The first pass stops at the last checkpoint end record.
- ii) The second pass starts at the last checkpoint start record with a matching checkpoint end record.
- iii) The third pass stops when the start record for all transactions to be undone have been seen.
- iv) The second pass stops at the end of the log.
- v) The first pass starts at the end of the log.

A) 0
B) 1
C) 2
D) 3
E) 4

ARIES Recovery Algorithm

Recovery algorithm described is a simplification of the ***ARIES*** recovery algorithm that is widely used in databases.

Three steps:

- 1) Analysis – determine dirty pages in buffer, active transactions, and starting point for REDO step
- 2) REDO – reapplies updates of committed transactions
- 3) UNDO – scan log backwards undoing updates for non-committed transactions

Implementation details:

- Every log record has a log sequence number (LSN).
- Also stores Transaction Table and Dirty Page Table.
- Handles failure during recovery by logging undo operations so do not have to be repeated (uses compensation log records).

Nonvolatile Storage Failures

Solution: Periodically **dump** the entire contents of the database to stable storage.

No transaction may be active during the dump procedure. A procedure similar to checkpointing must take place:

- Output all log records currently residing in main memory onto stable storage.
- Output all buffer blocks onto the disk.
- Copy the contents of the database to stable storage.
- Output a record <**dump**> to log on stable storage.

To recover from disk failure, restore database from most recent dump. Then log is consulted and all transactions that committed since the dump are redone.

- Can be extended to allow transactions to be active during dump; known as *fuzzy* or *online* dump.

Advanced Recovery Techniques

Support high-concurrency locking techniques, such as those used for B^+-tree concurrency control.

Operations like B^+-tree insertions and deletions release locks early. They cannot be undone by restoring old values (***physical undo***), since once a lock is released, other transactions may have updated the B^+-tree.

Instead, insertions/deletions are undone by executing a deletion/insertion operation (known as ***logical undo***).

- For such operations, undo log records should contain the undo operation to be executed; called ***logical undo logging***, in contrast to ***physical undo logging***.
- Redo information is logged *physically* (that is, new value for each write) even for such operations.

Undo/Redo Logging Questions

Explain undo/redo logging recovery for the following log as it appears at three instances of time:

(a)	(b)	(c)
<T_1 start>	<T_1 start>	<T_1 start>
<T_1, A, 4, 5>	<T_1, A, 4, 5>	<T_1, A, 4, 5>
<T_2 start>	<T_2 start>	<T_2 start>
<T_1 commit>	<T_1 commit>	<T_1 commit>
<T_2, B, 9, 10>	<T_2, B, 9, 10>	<T_2, B, 9, 10>
System Failure	<checkpoint start (T_2)>	<checkpoint start (T_2)>
	<T_2, C, 14, 15>	<T_2, C, 14, 15>
	<T_3 start>	<T_3 start>
	<T_3, D, 19, 20>	<T_3, D, 19, 20>
	System Failure	<checkpoint end>
		<T_2 commit>
		System Failure

Summary

A database system must be able to ***recover*** in the presence of hardware and software failures. The database system must ensure a consistent database after failure and preserve the ACID properties.

Log-based recovery records all updates in a log and undo/redo operations are used to restore the database to a consistent state (*write-ahead logging* is used).

Checkpointing reduces the cost of log-based recovery.

Database backups are needed to handle catastrophic failures.

Advanced (logical) recovery is necessary for B+-tree indexes.

Major Objectives

The "One Things":

- Perform Undo/Redo logging with checkpoints.

Major Theme:

- The recovery system rebuilds the database into a consistent state after failure using the log records saved to stable store while the database was operational. Various methods including checkpoints are used to speed-up recovery after failures.

Objectives

- Define: recovery and recovery system
- List the types of failures and motivation for recovery.
- Define: reliable, failure, error, fault, stable storage
- Compare/contrast fault avoidance versus fault tolerance.
- Read and write log records in a log.
- Define: write-ahead logging rule (WAL), log force operation
- Motivate the importance of checkpoints and online checkpointing.
- Compare/contrast physical versus logical logging.